Addis Abeba University
Addis Abeba Institute of Technology
School of Information Technology Engineering

Report on Parallel Programming: Assignment 1

Prepared By:
Elyas Abate

Nov 2024.

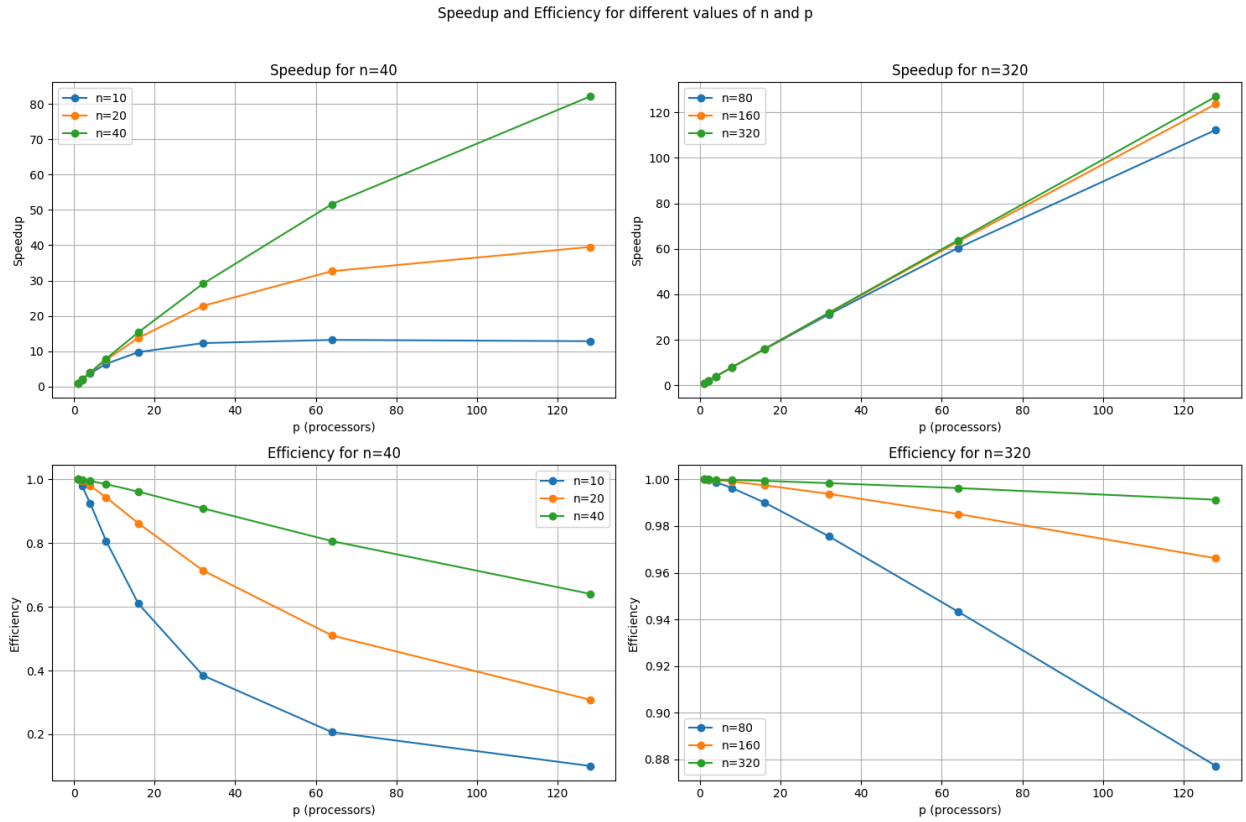
Submitted To:
D.r Beakal Gizachew

Problem 1

In this experiment, I examine the performance of a parallel program, focusing on the relationship between speedup and efficiency for different problem sizes (n) and varying numbers of processors (p). The serial runtime of the program is defined as $T_{serail} = n^2$, and the parallel runtime is modeled as $T_{parallel} = n^2/p + \log_2(p)$.

Part (a): Speedup and Efficiency Analysis

The performance of the parallelized program by calculating speedup and efficiency for various values of n (10, 20, 40, 80, 160, 320) and p (1, 2, 4, 8, 16, 32, 64, 128). The results are summarized as follows:



Observations:

1. As p is increased and n is held fixed:
    - For small values of n (e.g., n = 10), increasing the number of processors leads to diminishing returns in speedup. For example, speedup increases from 1 (at p = 1) to 13.2 (at p = 64), but further increasing p to 128 results in a lower speedup of 12.85. Efficiency also drops significantly with increasing p, from nearly 100% (at p = 1, 2) to 10% (at p = 128).

- For larger values of n (e.g., n = 320), the speedup continues to grow steadily as p increases, reaching nearly linear speedup. Efficiency also remains high, exceeding 99% for most values of p, even at p = 128.

2. As n is increased and p is fixed:
    - When p is fixed, the speedup and efficiency increase as n becomes larger. For instance, at p = 8, the speedup increases from 6.45 (at n = 10) to 7.99 (at n = 320), while the efficiency increases from 80.65% to 99.98%.
    - This indicates that larger problem sizes are more efficient in utilizing parallel resources, and the overhead becomes less significant as the computational load increases.

Part (b): Impact of Overhead on Efficiency

In this section, I analyze the effect of parallel overhead on efficiency as the problem size grows.

1. If the overhead grows more slowly than the serial time ($T_{overhead} < T_{serial}$):
    - As n increases, the ratio of the overhead to the total computation decreases, leading to an improvement in efficiency. The parallel efficiency approaches 100% as the problem size becomes large, as seen in the results for n = 320, where the efficiency for p = 128 is 99.13%.
    - This is because the time spent on overhead (e.g., synchronization, communication) becomes negligible compared to the time spent on useful computation, allowing the parallel system to perform closer to its ideal speedup.
2. If the overhead grows faster than the serial time $T_{overhead} > T_{serialasn} \rightarrow \infty$:
    - In this case, as n increases, the overhead dominates the runtime, causing the efficiency to decrease. For smaller values of n, especially with larger values of p, the overhead is significant relative to the problem size, resulting in reduced efficiency. For example, for n = 10 and p = 128, the efficiency is only 10.04%.
    - This illustrates the limitation of parallel systems when the overhead increases at a faster rate than the useful computational work, leading to diminishing returns on performance improvements.

Conclusion

From the analysis, observe that parallel efficiency increases as the problem size grows, provided the overhead does not scale faster than the serial computation time. This trend highlights the importance of balancing the problem size with the number of processors to achieve optimal parallel performance. When the problem size is too small, or the number of processors is too large, the parallel overhead can dominate, reducing the overall efficiency.

Problem 2

The tasked with developing a performance model for three versions (2, 3, and 5) of a parallel sum algorithm based on the given parameters:

- S: Sequential execution time.
- T: Number of threads.
- O: Overhead associated with parallelization.
- B: Barrier cost.
- M: Mutex cost per invocation.
- N: Number of elements being summed.

Execution Time of Parallel Versions

Version 2: Parallel Sum with Locks (Mutex)

In Version 2, each thread computes its portion of the sum and uses a mutex to ensure atomic updates to the global sum. Since each thread locks and unlocks the mutex each time it adds an element to the sum, there is overhead due to mutual exclusion.

Execution time:

- Each thread processes approximately N/T elements.
- For each element, there is a mutex lock and unlock, which adds a cost of M per element.
- The total execution time can be modeled as:
  $T_{version2} = S/T + O + N/T \cdot M$

Version 3: Parallel Sum with Reduced Locks

In Version 3, each thread computes its local sum independently, and the global sum is updated only once per thread, which reduces the frequency of mutex invocations compared to Version 2.

Execution time:

- Each thread processes N/T elements without locking.
- There is one mutex update per thread to add the local sum to the global sum, so the cost of locking is incurred T times.
- The total execution time can be modeled as:
  $T_{version3} = S/T + O + T \cdot M$.

Version 5: Parallel Sum with Barrier Synchronization

In Version 5, each thread computes its local sum, and a barrier is introduced to synchronize all threads before one thread (thread 0) computes the final global sum. The overhead of the barrier is B, and thread 0 has additional work to accumulate the local sums.

Execution time:

- Each thread processes N/T elements.
- After all threads finish their local sums, the barrier synchronization happens, which incurs the cost B.
- Thread 0 then accumulates the partial sums from all other threads, which takes an additional T steps.
- The total execution time can be modeled as:
  $T_{version5} = S/T + O + B + T$

Parallelization is Profitable for Version 3

For parallelization to be profitable, the parallel execution time should be less than the sequential execution time. This means:

$$T_{version3} < S$$

Substitute the expression for Tversion 3T_{\text{version 3}}$T_{version 3}$:

$$S/T + O + T \cdot M < S$$

Rearrange the inequality:

$$O + T \cdot M < S - S / T$$

Simplify:

$$O + T \cdot M < S \cdot (1 - 1/T)$$

Thus, parallelization is profitable when:

$$O + T \cdot M < S \cdot (T - 1) / T$$

This equation defines the condition under which Version 3 provides a performance benefit over the sequential execution. Specifically, the overhead O and the total cost of mutex updates must be less than the fraction of the sequential execution time that is reduced by parallelization.

Problem 3

stream benchmark with varying numbers of OpenMP threads (2, 4, and 16). This benchmark measures memory bandwidth by performing simple computational kernels, including copy, scale, add, and triad operations.

Experiment Setup and System Specifications:

- Processor: 12th Gen Intel(R) Core(TM) i7-1255U
- Memory: 3.7 GB
- Operating System: Linux 5.15.167.4-microsoft-standard-WSL2

Performance Analysis

The STREAM benchmark was executed with varying numbers of OpenMP threads to measure memory bandwidth performance. The results are summarized in the table below:

Number of Threads	Kernel	Best Rate (MB/s)	Avg Time (s)	Min Time (s)	Max Time (s)
2	Copy	18763.8	0.008840	0.008527	0.009604
2	Scale	17687.2	0.009786	0.009046	0.011147
2	Add	22781.2	0.011769	0.010535	0.014048
2	Triad	21967.9	0.011646	0.010925	0.012573
4	Copy	18930.6	0.009262	0.008452	0.011538
4	Scale	19095.4	0.009299	0.008379	0.010298
4	Add	21447.4	0.011887	0.011190	0.013437
4	Triad	22081.1	0.011331	0.010869	0.011992
16	Copy	17701.2	0.010576	0.009039	0.017319
16	Scale	17695.2	0.009821	0.009042	0.011971
16	Add	20331.9	0.012883	0.011804	0.015205
16	Triad	20290.9	0.013114	0.011828	0.016355

Observations and Analysis

- Increasing Threads: Generally, increasing the number of threads improves performance, especially for the Copy and Scale kernels. However, the performance gains diminish as the number of threads increases, particularly for the Add and Triad kernels.
- Memory Bandwidth Saturation: The system appears to be memory-bandwidth limited, as increasing the number of threads beyond a certain point does not yield significant performance improvements.
- Kernel Performance: The Triad kernel consistently shows the highest performance, followed by the Add kernel. The Copy and Scale kernels have slightly lower performance.
- Thread Overhead: The overhead associated with managing multiple threads might be impacting performance, especially when the number of threads is high.

Conclusion

The STREAM benchmark provides valuable insights into the memory bandwidth performance of a system. By analyzing the results, it is possible to identify potential performance bottlenecks and optimize the code to achieve better performance. However, it's important to consider the specific system configuration and workload when drawing conclusions.

Question 4

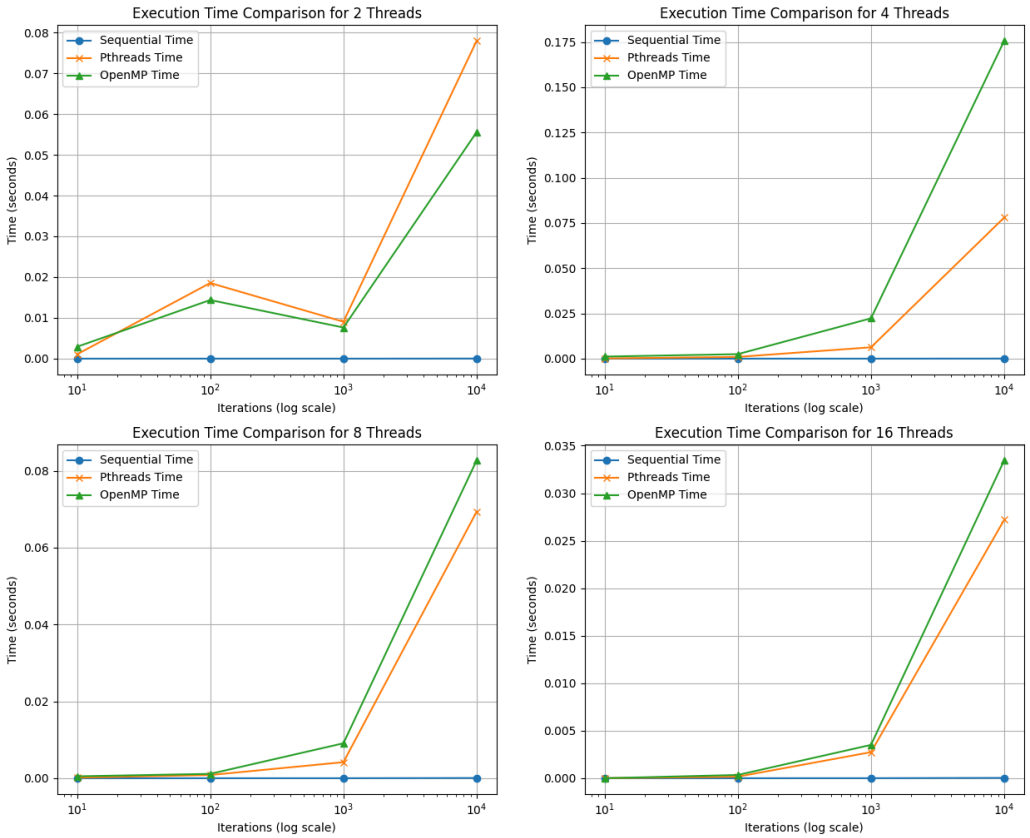
The aim of this experiment is to evaluate the performance of a serial program against two parallel implementations (Pthreads and OpenMP) for different thread counts, using varying problem sizes. The code was run 10 times: the first five trials for a smaller problem size (10,000 iterations) and the next five trials for a significantly larger problem size (10,000,000,000 iterations). The purpose of this was to demonstrate that while a serial program might perform well for smaller datasets, parallelization with Pthreads and OpenMP shows clear advantages as the problem size increases.

Experimental Setup

- Serial program: The sequential version of the algorithm.
- Pthreads implementation: Uses POSIX threads for parallel computation.
- OpenMP implementation: Uses OpenMP for thread-based parallelism.

The performance for four thread counts (2, 4, 8, and 16 threads) and different iteration counts ranging from 10 to 10,000,000,000. The results were averaged over five trials for both small and large input sizes.

Results



Small Problem Size (10,000 iterations)

Iterations	Threads	Sequential Time (s)	Pthreads Time (s)	OpenMP Time (s)
10	2	0.000001	0.002571	0.008909
10	4	0.000000	0.000000	0.000000
…	..	..	..	..
10,000	4	0.000034	0.041377	0.153598
10,000	8	0.000071	0.056886	0.025946
10,000	16	0.000042	0.040978	0.035081

Large Problem Size (10,000,000,000 iterations)

Iterations	Threads	Sequential Time (s)	Pthreads Time (s)	OpenMP Time (s)
10,000	2	0.000023	0.123711	0.070640
10,000	4	0.000030	0.091904	0.210000
10,000	8	0.000030	0.045242	0.120574
10,000	16	0.000029	0.004052	0.025695
–	–	–	–	–
10,000,000,000	2	10.769920	1.783349	1.813161
10,000,000,000	4	10.287990	1.828434	2.142222
10,000,000,000	8	10.014093	2.439986	2.471573
10,000,000,000	16	12.021655	4.210630	3.402520

Analysis

Small Problem Size (10,000 Iterations)

For smaller problem sizes (up to 10,000 iterations), the sequential program performed exceptionally well. The Pthreads and OpenMP implementations had overheads due to thread creation and management, leading to significantly higher execution times compared to the sequential version.

For instance, with 10,000 iterations:

- Sequential execution took only 0.000029 seconds.
- The Pthreads implementation with 2 threads took 0.097992 seconds, which is substantially slower.
- OpenMP also showed similar behavior, with 0.057358 seconds for 2 threads.

This indicates that for smaller datasets, the cost of parallelization outweighs the benefits, and a sequential approach is more efficient.

Large Problem Size (10,000,000,000 Iterations)

For large datasets, the performance of the parallel implementations, both Pthreads and OpenMP, improves significantly compared to the sequential program. As the problem size increases:

- Pthreads and OpenMP perform much better due to their ability to distribute work across multiple threads.
- For instance, with 10,000,000,000 iterations:

- Sequential execution took 10.769920 seconds.
- With 16 threads, Pthreads took 4.210630 seconds, while OpenMP took 3.402520 seconds, which is a clear performance gain.

Observations

1. Thread count impact: Increasing the number of threads generally improves performance for larger datasets. However, the overhead of thread management becomes noticeable for smaller inputs.
2. Pthreads vs. OpenMP: Both parallel implementations showed significant speedup for larger problem sizes. OpenMP performed slightly better in some cases, especially as the number of iterations increased.
3. Parallelization overhead: For small problem sizes, the overhead introduced by parallelization mechanisms (Pthreads/OpenMP)